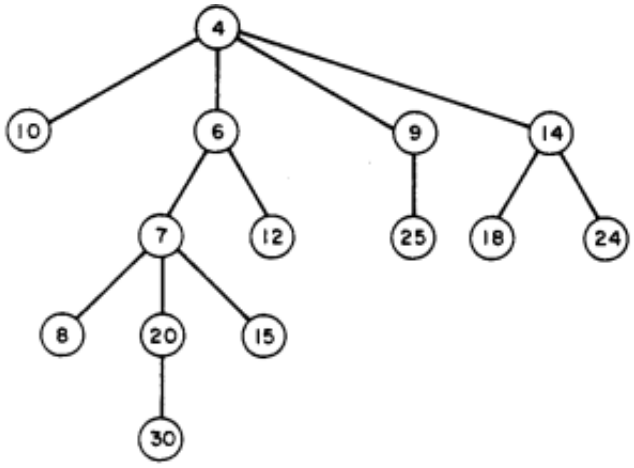
Haskell (IR3) – TP noté

Stéphane Vialette (stephane.vialette@univ-eiffel.fr)

13 janvier 2023

Nous considérons dans la suite des arbres étiquettés d'arité variable. Nous parlons d'*arbre étiquetté* car chaque nœud de l'abre porte une valeur et d'*arité variable* car chaque nœud possède un nombre quelconque de fils (y compris 0 si c'est une feuille).

Vous devez écrire toutes vos fonctions dans un fichier nommé `MTree.hs`.



Les types **`MTree`** (pour *Multiway Tree*) et **`MForest`** (pour *Multiway Forest*) sont donnés et doivent être utilisés tels quels dans la suite. Remarquez que **`MTree`** est un type *record* à deux champs/fonctions alors que **`MTree`** est un type *alias*.

```haskell
import qualified Data.Foldable as F
import qualified Data.List     as L

data MTree a = MTree {rootLabel :: a, subForest :: MForest a}
               deriving (Eq, Ord)

type MForest a = [MTree a]

mkMTree :: a -> [MTree a] -> MTree a
mkMTree rl ts = MTree {rootLabel = rl, subForest = ts}

mkMTreeLeaf :: a -> MTree a
mkMTreeLeaf rl = mkMTree rl []
```

```haskell
mTreeExample :: MTree Integer
mTreeExample = root
  where
    root = mt04
      where
        mt04 = mkMTree      4 [mt10, mt06, mt09, mt14]
        mt10 = mkMTreeLeaf 10
        mt06 = mkMTree      6 [mt07, mt12]
        mt07 = mkMTree      7 [mt08, mt20, mt15]
        mt08 = mkMTreeLeaf  8
        mt20 = mkMTree     20 [mt30]
        mt30 = mkMTreeLeaf 30
        mt15 = mkMTreeLeaf 15
        mt12 = mkMTreeLeaf 12
        mt09 = mkMTree      9 [mt25]
        mt25 = mkMTreeLeaf 25
        mt14 = mkMTree     14 [mt18, mt24]
        mt18 = mkMTreeLeaf 18
        mt24 = mkMTreeLeaf 24
```

FIGURE 1 – Un arbre.

L'instance de la classe `Show` est également donnée pour ces arbres. À titre d'illustration, l'arbre défini en FIGURE 1 (`mTreeExample :: MTree Integer` qui produit l'arbre dessiné plus haut) s'affiche comme montré en FIGURE 2.

```haskell
instance (Show a) => Show (MTree a) where
  show = go 0
    where
      go nTabs MTree {rootLabel = rl, subForest = mts} =
        L.replicate nTabs '-'           ++
        (if nTabs > 0 then " " else "") ++
        "+ root label="                 ++
        show rl                         ++
        "\n"                            ++
        F.foldr f "" mts
        where
          f mt acc = go (nTabs + 4) mt ++ acc
```

```
+ root label=4
.... + root label=10
.... + root label=6
........ + root label=7
............ + root label=8
............ + root label=20
.............. + root label=30
............ + root label=15
........ + root label=12
.... + root label=9
........ + root label=25
.... + root label=14
........ + root label=18
........ + root label=24
```

FIGURE 2 – Visualisation de l'arbre `mTreeExample`.

Question 1: Quelques fonctions élémentaires

(a) Écrire la fonction

```
mTreeIsLeaf :: MTree a -> Bool
```

qui retroune vrai si un arbre est une feuille.

```
λ: mTreeIsLeaf mTreeExample
False
λ: let mt = mkMTreeLeaf "some dummy data" in mTreeIsLeaf mt
True
```

(b) Écrire la fonction

```
mTreeCount :: Num b => MTree a -> b
```

qui calcule le nombre de nœuds d'un arbre.

```
λ: mTreeCount mTreeExample
14
```

(c) Écrire la fonction

```
mTreeCountLeaves :: Num b => MTree a -> b
```

qui calcule le nombre de feuilles d'un arbre.

```
λ: mTreeCountLeaves mTreeExample
8
```

(d) Écrire la fonction

```
mTreeHeight :: (Num b, Ord b) => MTree a -> b
```

qui calcule la hauteur d'un arbre.

```
λ: mTreeHeight mTreeExample
5
```

Question 2: Calculs sur les `MTree` et fonctions d'ordre supérieur

(a) Écrire la fonction

```
mTreeMin :: Ord a => MTree a -> a
```

qui retourne l'étiquette minimale d'un arbre.

```
λ: mTreeMin mTreeExample
4
```

(b) Écrire la fonction

```
mTreeMax :: Ord a => MTree a -> a
```

qui retourne l'étiquette maximale d'un arbre.

```
λ: mTreeMax mTreeExample
30
```

(c) Écrire la fonction

```
mTreeSum :: Num a => MTree a -> a
```

qui calcule la somme des étiquettes d'un arbre.

```
λ: mTreeSum mTreeExample
202
```

(d) Écrire la fonction d'ordre supérieur

```
mTreeMap :: (a -> b) -> MTree a -> MTree b
```

qui calcule un nouvel arbre en appliquant une fonction donnée sur toutes les étiquettes d'une arbre.

```
λ: mTreeMap (* 100) mTreeExample
+ root label=400
.... + root label=1000
.... + root label=600
........ + root label=700
............ + root label=800
............ + root label=2000
................ + root label=3000
............ + root label=1500
........ + root label=1200
.... + root label=900
........ + root label=2500
.... + root label=1400
........ + root label=1800
........ + root label=2400
λ: mTreeMap even mTreeExample
+ root label=True
.... + root label=True
.... + root label=True
........ + root label=False
............ + root label=True
............ + root label=True
................ + root label=True
............ + root label=False
........ + root label=True
.... + root label=False
........ + root label=False
.... + root label=True
........ + root label=True
........ + root label=True
```

(e) Écrire la fonction d'ordre supérieur

```
mTreeFilter :: (a -> Bool) -> MTree a -> MForest a
```

qui calcule un nouvel arbre obtenu par filtrage des étiquettes d'un arbre. Remarquons que la fonction retourne une forêt contenant un unique arbre si et seulement si la racine de l'arbre est conservée par le filtrage.

```
λ: mTreeFilter even mTreeExample
[+ root label=4
.... + root label=10
.... + root label=6
........ + root label=8
........ + root label=20
............ + root label=30
........ + root label=12
.... + root label=14
........ + root label=18
........ + root label=24
]
λ: mTreeFilter odd mTreeExample
[+ root label=7
.... + root label=15
,+ root label=9
.... + root label=25
]
λ: mTreeFilter (< 10) mTreeExample
[+ root label=4
.... + root label=6
........ + root label=7
............ + root label=8
.... + root label=9
]
λ: mTreeFilter (\x -> even x && x `mod` 3 == 0) mTreeExample
[+ root label=6
.... + root label=30
.... + root label=12
,+ root label=18
,+ root label=24
]
```

Question 3: Collectes et parcours

(a) Écrire la fonction

```
mTreeLeaves :: MTree a -> [a]
```

qui calcule la liste des étiquettes des feuilles d'un arbre. Aucun ordre spécifique n'est imposé.

```
λ: mTreeLeaves mTreeExample
[10,8,30,15,12,25,18,24]
```

(b) Écrire la fonction

```
mTreeFlatten :: MTree a -> [a]
```

qui calcule la liste de toutes les étiquettes d'un arbre. Aucun ordre spécifique n'est imposé.

```
λ: mTreeFlatten mTreeExample
[4,10,6,7,8,20,30,15,12,9,25,14,18,24]
```

(c) Écrire la fonction

```
mTreeBreadthFirstTraversal :: MTree a -> [a]
```

qui calcule la liste de toutes les étiquettes d'un arbre rencontrées dans un parcours en largeur.

```
λ: mTreeBreadthFirstTraversal  mTreeExample
[4,10,6,9,14,7,12,25,18,24,8,20,15,30]
```

(d) Écrire la fonction

```
mTreeDepthFirstTraversal :: MTree a -> [a]
```

qui calcule la liste de toutes les étiquettes d'un arbre rencontrées dans un parcours en profondeur.

```
λ: mTreeDepthFirstTraversal mTreeExample
[4,10,6,7,8,20,30,15,12,9,25,14,18,24]
```

(e) Écrire la fonction

```
mTreeLayer :: Int -> MTree a -> [a]
```

qui calcule la liste de toutes les étiquettes d'un arbre à une profondeur donnée (la racine est à la hauteur 1). Aucun ordre spécifique n'est imposé.

```
λ: mapM_ print [("layer " ++ show l, mTreeLayer l mTreeExample) | l <- [0..5]]
("layer 0",[])
("layer 1",[4])
("layer 2",[10,6,9,14])
("layer 3",[7,12,25,18,24])
("layer 4",[8,20,15])
("layer 5",[30])
```

Question 4: Programmation origami (1/2)

Considérons la fonction d'ordre supérieur `mTreeFold` donnée ci-après :

```
mTreeFold :: (a -> [b] -> b) -> MTree a -> b
mTreeFold f t = f rl (L.map (mTreeFold f) ts)
  where
    rl = rootLabel t
    ts = subForest t
```

(a) En utilisant exclusivement la fonction `mTreeFold`, écrire la fonction

```
mTreeMin' :: Ord a => MTree a -> a
```

qui retourne l'étiquette minimale d'un arbre.

```
λ: mTreeMin' mTreeExample
4
```

(b) En utilisant exclusivement la fonction `mTreeFold`, écrire la fonction

```
mTreeMax' :: Ord a => MTree a -> a
```

qui retourne l'étiquette maximale d'un arbre.

```
λ: mTreeMax' mTreeExample
30
```

(c) En utilisant exclusivement la fonction `mTreeFold`, écrire la fonction

```
mTreeFlatten' :: MTree a -> [a]
```

qui calcule la liste de toutes les étiquettes d'un arbre.

```
λ: mTreeFlatten' mTreeExample
[4,10,6,7,8,20,30,15,12,9,25,14,18,24]
```

Question 5: Programmation origami (2/2)

Considérons maintenant la fonction d'ordre supérieur `mTreeFold'` :

```haskell
mTreeFold' :: (a -> b -> b) -> b -> MTree a -> [b]
mTreeFold' f z MTree {rootLabel = rl, subForest = []} = [f rl z]
mTreeFold' f z t = [f rl y | t' <- ts, y <- mTreeFold' f z t']
  where
    rl = rootLabel t
    ts = subForest t
```

(a) En utilisant exclusivement la fonction `mTreeFold'`, écrire la fonction

`mTreeCollectPaths :: MTree a -> [[a]]`

qui calcule toutes les listes énumérant les étiquettes depuis la racine jusqu'à une feuille.

```
λ: mapM_ print (mTreeCollectPaths mTreeExample)
[4,10]
[4,6,7,8]
[4,6,7,20,30]
[4,6,7,15]
[4,6,12]
[4,9,25]
[4,14,18]
[4,14,24]
```

Question 6: Génération

Il s'agit dans cette dernière partie d'écrire la fonction

`mTrees :: (Eq a) => [a] -> [MTree a]`

permettant de générer tous les arbres distincts portant un même ensemble d'étiquettes.

```
λ: mTrees []
[]
λ: mTrees [1]
[+ root label=1
]
λ: mTrees [1,2,3]
[+ root label=1
.... + root label=2
.... + root label=3
,+ root label=1
.... + root label=2
........ + root label=3
,+ root label=1
.... + root label=3
.... + root label=2
,+ root label=1
.... + root label=3
........ + root label=2
,+ root label=2
.... + root label=1
.... + root label=3
,+ root label=2
.... + root label=1
........ + root label=3
,+ root label=2
.... + root label=3
```

```
.... + root label=1
,+ root label=2
.... + root label=3
........ + root label=1
,+ root label=3
.... + root label=1
.... + root label=2
,+ root label=3
.... + root label=1
........ + root label=2
,+ root label=3
.... + root label=2
.... + root label=1
,+ root label=3
.... + root label=2
........ + root label=1
]
λ: [(rls, L.length (mTrees rls)) | maxL <- [1..5], let rls = [1..maxL]]
[([1],1),([1,2],2),([1,2,3],12),([1,2,3,4],168),([1,2,3,4,5],6600)]
```

(a) Pour faciliter l'écriture de la fonction `mTrees`, écrire dans un premier temps la fonction

```
permutedSubsets :: [a] -> [[a]]
```

qui calcule toutes les permutations de tous les sous-ensembles d'une liste (y compris le sous-ensemble vide). Rappelons que le module `Data.List` fournit la fonction `permutations :: [a] -> [[a]]` qui calcule toutes les permutations d'une liste.

```
λ: mapM_ print (permutedSubsets [])
[]
λ: mapM_ print (permutedSubsets [1])
[]
[1]
λ:  mapM_ print (permutedSubsets [1..2])
[]
[2]
[1]
[1,2]
[2,1]
λ: mapM_ print (permutedSubsets [1..3])
[]
[3]
[2]
[2,3]
[3,2]
[1]
[1,3]
[3,1]
[1,2]
[2,1]
[1,2,3]
[2,1,3]
[3,2,1]
[2,3,1]
[3,1,2]
[1,3,2]
```

(b) Supposons que la fonction `mTrees` soit définie de la façon suivante :

```
mTrees :: (Eq a) => [a] -> MForest a
mTrees rls = [mkMTree rl ts | rl <- rls
                            , ts <- mForests (L.delete rl rls)]
```

Écrire maintenant la fonction

`mForests :: (Eq a) => [a] -> [MForest a]`

afin d'obtenir le calcul souhaité pour la fonction `mTrees`.